Clique

→ largest SAV that forms a complete graph

→ Find maximal by sorting according to ?

→ Find a dense subgraph instead

Vertex Coloring

→ 2-coloring: it's covering bipartite components → DFS.

→ Finding chromatic number of graph is NP complete.

Edge Coloring

→ Vizing's theorem: if Δ max deg, we can edge color with Δ+1.

↳ This offers an O(|V||E|) algorithm for finding (Δ+1)-coloring of the graph

Text Compression

- Can it be lossy?
- Can I simplify? (Burrows-Wheeler)
- Does it have to be real-time?

→ Huffman codes: ≥ two passes if not adaptive

→ Lempel-Ziv (including LZW): build index as we go and leave frequent words.

Programming Pearls $\boxed{2}^{nd}$

Jon Bentley

col 1 - col 15

8 pages

2016,11,14-2016,11,16

①

- state the problem in clear terms
- use a bit-vector to represent a set of numbers in a given range. ⇒ Bitmap Data Structure

↳ We can sort such a file with a very limited number of passes.

②

- Use binary search (one/two sided) wherever possible.

↳ binary search of the problem state doesn't have to be about finding an item in a sorted list ⇒ find the missing item

- rotating vector $S_{1..n}$ at i to: $S_{i..n} S_{1..i}$

↳ ① Use shifts

② Use pointers

③ Use half jumps: $\begin{cases} S_i \Rightarrow S_0 \\ S_{2i} \Rightarrow S_i \\ \dots \end{cases}$

④ See $ab \Rightarrow ba$ as $\begin{cases} a b_l b_r \\ |a| = |b_r| \end{cases}$ then swap a for b_r ⇒ $\underline{b_r b_l} a$ and recurse

⑤ Reverse: $ab \rightarrow a^r b$
$a^r b \rightarrow a^r b^r$
$a^r b^r \rightarrow (a^r b^r)^r = ba$

→ Find anagrams ⇒ reduce words to their signature by lexicographically sorting each word (hello → ehllo) and finding words with the same signature.

- Good programmers are a little bit lazy: they sit back and think before rushing to code.

---
③

→ Use templating and separate control from results.

→ Extract switch parameters into a DB

⇒ In general think of extensibility.

- Use "Data Structures" to reduce big programs to small programs.

- Generalization helps form an objective view.

⟹ Let the data, structure the program!

---
④

- Be liberal with assertions
- Control loops at all three stages: see CLRS(3)
- Clearly define the contract: ① pre-conditions ② post-conditions
- Assertion provide insight into intentions
- Try to program units simple and small.

(5)

- Instead of directly incorporating the new code into the system build a scaffolding.
- run the function in isolation against easily verifiable test cases.
- put meaningful assertions in the code that do not cause errors themselves
- write automated tests that exhaustively runs our code through possible valid AND invalid inputs.
- We can also test run time complexity by measuring actual runtime and check if it falls within ε of expected time.

(6)

- Efficiency can be achieved at various levels.
  - problem specification
  - system structure & modularization
  - algorithms & data structures
  - code tuning
  - system software
  - hardware

→ figure out which gives you the biggest boost with the least effort & possibly work on many levels

---

- Try back-of-the-envelope estimates
  ↳ try your calculations from multiple points of view if possible
- Check the units, check the obvious issues with your basic calculations.

▶ π seconds is a nano century.

- Try to estimate performance and space efficiency in this way.
- Incorporate safety factors to compensate for mistakes & oversimplifications.
- Little's Law: calculate entry rate based on exit rate and time spent in-the-process

---

- Save the state to avoid recomputation
- Preprocess data to help with the actual processing
- Try to see if solving for $n/2$ helps solving for n → divide & conquer
- See if the solution for $x[0..i-1]$ can be extended for $x[i]$ → scanning technique
- Use cummulative data when dealing with ranges.
- Find out the lower bounds!

9

- Premature optimization is the root of ~~all~~ many :D evil.
- use measurement tools to zoom in on the culprit.
- make sure your code optimizations to the small scale do not affect the overall speed
- try to follow these rules in code optimization
  - exploit algebraic identities
  - collapse procedure hierarchies
  - unroll the loops
  - augment the data structures

10

- Simplicity of data structures is usually the key to reducing data / memory usage
- Be aware of the density / sparsity of data
- reducing data space → recompute
  - ← use dynamic allocation
  - ← use garbage collection
  - → exploit sparsity
  - → use compression techniques
  - → use pointers to shared space
- reducing code space → Factor into functions
  - ← write machine code :-(
  - → use interpreters
  - → optimize the compiler

(11)

- Do not implement generic algorithms such as sorting unless you have reason to.
- Sorting is a powerful tool that might be overused at times.

---

(12)

▲ An important part of a programmer's job is solving tomorrow's problems:

→ understand the perceived problem & do not take the implied solution as the way to go.

→ abstract the problem to see how it relates to other problems.

→ use an informal high-level language to design a solution then evaluate the merits of different algorithms/data structures <u>before coding</u>.

{ - implement a solution and optimize it
{ - prototype many solutions & compare them

→ do retrospections after the solution to see how ~~F~~ else/better you could've done

(13)

- Using interfaces to decouple the contract from implementation let's us improve existing code by dropping in better implementations
- consider using libraries in place of home made implementations unless absolutely necessary.
- remember that not all space is equal. Know when your data is crossing from external storage to main memory, to CPU cache and within RAM fragments
- Use code tuning (see 9) to optimize

(14)

- always measure the efficiency of code
- stating loop invariants helps with validating correctness
- distinguish between "what" & "how" (interface, implementation)
- use abstraction to allow for easy fixes and improvements ← decouple what & how

⇒ heap sort can make use of freed heap space to co-host both the sorted array & the heap

- Suffix arrays are arrays of proper suffixes of a given string:

SAMPLE : A_0 SAMPLE
A_1 AMPLE
A_2 MPLE
A_3 PLE
A_4 LE
A_5 E

We can sort them lexicographically. after that comparing two adjacent entries' prefixes can yield the longest repeated substring of S.

- Generating random text & searching for phrases are among the uses of suffix arrays.

Introduction to Information Retrieval

Christopher D. Manning

Prabhakar Raghavan

ch 1 — ch 21

taky @ PKU CS

2016.11.17 - 2016.11.28